IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

APPLICATION FOR LETTERS PATENT

SERVICE DISCOVERY AND PUBLICATION

Inventor(s):
Jeffrey B Parham
Charles R. Reeves
Andrew D. Milligan
Gopal Krishna R. Kakivaya
Lawrence A. Buerk
Angela Mills
Richard L Hasha

ATTORNEY'S DOCKET NO. MS1-1802US
CLIENT'S DOCKET NO. 306297.01

SERVICE DISCOVERY AND PUBLICATION

TECHNICAL FIELD

[0001] The described subject matter relates to digital computing, and more particularly to service discovery in computing devices and computing networks.

BACKGROUND

[0002] Application programs that execute on computing devices and computer networks may require the use of services provided by other physical or logical devices connected to the computing device or network. Presently, application programs use a wide range of application programming interfaces (APIs), protocols, and object models to discover, enumerate, and describe services and devices on a local computing device or across a plurality of devices in a computer network. The mechanisms available to discover, enumerate, and describe services and devices differ significantly, even when the services and devices involved are conceptually similar.

[0003] For example, consider a situation in which an application seeks to enumerate available printers. When executing within an administered, corporate environment, the application may need to use Lightweight Directory Access Protocol (LDAP) to communicate with a Microsoft Active Directory® directory service store to discover registered corporate printers, NetBT to discover print queue servers, and Bluetooth to discover personal area network printers. In addition, the application might have to invoke device management APIs to

discover direct attached printers, and UPnP™ APIs to discover UPnP printers. Each of these mechanisms requires understanding of a particular API, protocol, and query semantic.

[0004] The number of APIs and protocols required to for an application to discover, enumerate, and describe services complicates the task of software development.

SUMMARY

[0005] Implementations described and claimed herein address these and other problems by providing a uniform interface that simplifies discovery and publication tasks. The uniform interface permits underlying protocols to be leveraged and eliminates the need for application developers to understand low-level protocols. The uniform interface provides a consistent, high-level abstraction of services and associated operations that targets the discovery and publication of service details over a wide range of lower-level APIs, protocols, stores, and network environments.

[0006] In one exemplary implementation, a method for discovering services available in a computing environment is provided. The method comprises: in an application program, defining a discovery scope; defining a discovery filter; and initiating a search request to a first application programming interface; and in the first application programming interface: parsing the search request; retrieving

service information corresponding to the requested discovery scope and discovery filter; and returning the service information to the application program.

[0007] In another exemplary implementation, a method for publishing services available in a computing environment is provided. The method comprises, in an application program: defining a service entry object; defining a publication scope; assigning a unique key to the service; assigning a service type; defining properties for the service; and defining endpoints for the service; and initiating a publication request to a first application programming interface; and in the first application programming interface: parsing the search request; and executing at least one low-level API call to publish the service.

[0008] In another exemplary implementation, a method for deleting a published service in a computing environment is provided. The method comprises, in an application program: defining a service entry object; specifying a key corresponding to the published service; defining a deletion scope; and initiating a deletion request to a first application programming interface; and in the first application programming interface: parsing the search request; and executing at least one low-level API call to delete the service.

[0009] In another exemplary implementation, a method of subscribing to service events in a computing environment is provided. The method comprises, in an application program: defining a scope; defining a filter; defining a callback function; and initiating a subscription request to a first application programming

interface; and in the first application programming interface: parsing the search request; and executing at least one low-level API call to subscribe to service events; and returning information from service events to the application program.

[0010] In another exemplary implementation, a system for managing information about services available in a computing environment is provided. The system comprises a first application programming interface configured to accept service queries from an application, wherein the first application programming interface receives service queries in a first service query protocol, processes the service queries, and launches at least one corresponding service query to a second protocol; a discovery persistence service communicatively connected to the first application programming interface, wherein the discovery persistence service receives service information from the first application programming interface and stores the service information in a data store.

BRIEF DESCRIPTION OF THE DRAWINGS

[0011] Fig. 1 is a schematic illustration of an exemplary computing device;

[0012] Fig. 2 is a block diagram illustrating an exemplary software architecture;

[0013] Fig. 3 is a flowchart illustrating operations for service discovery;

[0014] Fig. 4 is a flowchart illustrating operations for service publication;

[0015] Fig. 5 is a flowchart illustrating operations for service deletion;

[0016] Fig. 6 is a flowchart illustrating operations for subscribing to service events;

[0017] Fig. 7 is a block diagram illustrating the relationship between concrete scopes and abstract scopes

[0018] Fig. 8 is pseudo-code illustrating how to use the C# programming language to locate color printers that print 50 pages per minute using a SimpleFilter object on the Active Directory protocol;

[0019] Fig. 9 is pseudo-code illustrating how to use the C# programming language to locate Web services;

[0020] Fig. 10 is pseudo-code illustrating the use of the C# programming language to find services supporting a specific tModel interface using a SimpleFilter object and the UDDI protocol;

[0021] Fig. 11 is pseudo-code illustrating the use of Visual Basic.NET to find services supporting a specific tModel interface using a SimpleFilter object and the UDDI protocol;

[0022] Fig. 12 is pseudo-code illustrating the use of the C# programming language to locate a printer with a name like Office Printer using the RichFilter with Active Directory;

[0023] Fig. 13 is pseudo-code illustrating the use of Visual Basic.NET to locate a printer with a name like Office Printer using the RichFilter with Active Directory;

[0024] Fig. 14 is pseudo-code illustrating the use of the C# programming language to publish a service of a specific type, identified by a specific unique identifier, using the SSDP protocol;

[0025] Fig. 15 is pseudo-code illustrating the use of Visual Basic.NET to publish a service of a specific type, identified by a specific unique identifier, using the SSDP protocol;

[0026] Fig. 16 is pseudo-code illustrating the use of the C# programming language to delete a service from the SSDP protocol;

[0027] Fig. 17 is pseudo-code illustrating the use of Visual Basic.NET to delete a service from the SSDP protocol;

[0028] Fig. 18 is pseudo-code illustrating the use of the C# programming language to use a SimpleFilter to register for events of a specific type that use the SSDP protocol. The registered callback function will be invoked for every event that matches the filter and the corresponding ServiceEntry object will be provided to that handler; and

[0029] Fig. 19 is pseudo-code illustrating the use of Visual Basic.NET to use a SimpleFilter to register for events of a specific type that use the SSDP protocol.

DETAILED DESCRIPTION

[0030] Described herein are exemplary methods and software architecture for service discovery and publication. The methods described herein may be embodied as logic instructions on a computer-readable medium. When executed on a processor, the logic instructions cause a general purpose computing device to be programmed as a special-purpose machine that implements the described methods. The processor, when configured by the logic instructions to execute the methods recited herein, constitutes structure for performing the described methods.

Exemplary Operating Environment

[0031] Fig. 1 is a schematic illustration of an exemplary computing device 130 that can be utilized to implement one or more computing devices in accordance with the described embodiment. Computing device 130 can be utilized to implement various implementations in accordance with described embodiments.

[0032] Computing device 130 includes one or more processors or processing units 132, a system memory 134, and a bus 136 that couples various system components including the system memory 134 to processors 132. The bus 136 represents one or more of any of several types of bus structures, including a memory bus or memory controller, a peripheral bus, an accelerated graphics port, and a processor or local bus using any of a variety of bus architectures. The system memory 134 includes read only memory (ROM) 138 and random access

memory (RAM) 140. A basic input/output system (BIOS) 142, containing the basic routines that help to transfer information between elements within computing device 130, such as during start-up, is stored in ROM 138.

[0033] Computing device 130 further includes a hard disk drive 144 for reading from and writing to a hard disk (not shown), a magnetic disk drive 146 for reading from and writing to a removable magnetic disk 148, and an optical disk drive 150 for reading from or writing to a removable optical disk 152 such as a CD ROM or other optical media. The hard disk drive 144, magnetic disk drive 146, and optical disk drive 150 are connected to the bus 136 by an SCSI interface 154 or some other appropriate interface. The drives and their associated computer-readable media provide nonvolatile storage of computer-readable instructions, data structures, program modules and other data for computing device 130. Although the exemplary environment described herein employs a hard disk, a removable magnetic disk 148 and a removable optical disk 152, it should be appreciated by those skilled in the art that other types of computer-readable media which can store data that is accessible by a computer, such as magnetic cassettes, flash memory cards, digital video disks, random access memories (RAMs), read only memories (ROMs), and the like, may also be used in the exemplary operating environment.

[0034] A number of program modules may be stored on the hard disk 144, magnetic disk 148, optical disk 152, ROM 138, or RAM 140, including an operating system 158, one or more application programs 160, other program

modules 162, and program data 164. A user may enter commands and information into computing device 130 through input devices such as a keyboard 166 and a pointing device 168. Other input devices (not shown) may include a microphone, joystick, game pad, satellite dish, scanner, or the like. These and other input devices are connected to the processing unit 132 through an interface 170 that is coupled to the bus 136. A monitor 172 or other type of display device is also connected to the bus 136 via an interface, such as a video adapter 174. In addition to the monitor, personal computers typically include other peripheral output devices (not shown) such as speakers and printers.

[0035] Computing device 130 commonly operates in a networked environment using logical connections to one or more remote computers, such as a remote computer 176. The remote computer 176 may be another personal computer, a server, a router, a network PC, a peer device or other common network node, and typically includes many or all of the elements described above relative to computing device 130, although only a memory storage device 178 has been illustrated in Fig. 1. The logical connections depicted in Fig. 1 include a local area network (LAN) 180 and a wide area network (WAN) 182. Such networking environments are commonplace in offices, enterprise-wide computer networks, intranets, and the Internet.

[0036] When used in a LAN networking environment, computing device 130 is connected to the local network 180 through a network interface or adapter

184. When used in a WAN networking environment, computing device 130 typically includes a modem 186 or other means for establishing communications over the wide area network 182, such as the Internet. The modem 186, which may be internal or external, is connected to the bus 136 via a serial port interface 156. In a networked environment, program modules depicted relative to the computing device 130, or portions thereof, may be stored in the remote memory storage device. It will be appreciated that the network connections shown are exemplary and other means of establishing a communications link between the computers may be used.

[0037] Generally, the data processors of computing device 130 are programmed by means of instructions stored at different times in the various computer-readable storage media of the computer. Programs and operating systems are typically distributed, for example, on floppy disks or CD-ROMs. From there, they are installed or loaded into the secondary memory of a computer. At execution, they are loaded at least partially into the computer's primary electronic memory. The invention described herein includes these and other various types of computer-readable storage media when such media contain instructions or programs for implementing the steps described below in conjunction with a microprocessor or other data processor. The invention also includes the computer itself when programmed according to the methods and techniques described below.

Exemplary Software Architecture Overview

[0038] **Fig. 2** is a block diagram of an exemplary software architecture 200 for service discovery that may reside in system memory 134 of Fig. 1. In this implementation, system memory 134 may comprise a plurality of application programs 210. In a networked environment the application programs may function as client programs, while in a PC environment the applications may execute as stand-alone programs. The particular nature of the application programs is not critical.

[0039] Application programs 210 invoke service discovery API 214 to discover services available in the computing environment. Service discovery API 214 provides a high-level grammar for expressing discovery queries. The grammar may be implemented in OPath, a natural query language used for expressing discovery queries. This high-level grammar provides software developers a more conceptual mechanism to express the service(s) the developer is looking for, rather than requiring a more granular and protocol-specific expression that may be required by the underlying protocols 220-234. The developer can construct a query using the high-level grammar, which may then be forwarded to either a specific set of protocols, referred to as a number of "concrete scopes", or use an "abstract scope" which is a predefined or configured set of concrete scopes.

In addition to supporting service discovery, the system supports service publication/deleting, and monitoring for events.

[0040] Service discovery API 214, in turn, invokes one or more underlying protocols, represented in the diagram by Protocol 1 220 through Protocol 8 234. The particular number of underlying protocols is not important. Certain of the protocols 220-234 may be directory-backed protocols such as, e.g., LDAP, Universal Description, Discovery and Integration (UDDI), and Domain Name System (DNS) Server. Other protocols may be ad-hoc protocols such as, e.g., Bluetooth, UPnP, and NetBT. One or more of the underlying protocols 220-234 uses a communication connection 236 to communicate with other components or services available in the computing environment.

[0041] In response to the discovery request, the service discovery API returns a collection of ServiceEntry objects that represent matching services discovered either on the local machine or on the network. A ServiceEntry object is a generalized data structure that can represent much of the relevant detail returned by the underlying protocols that system supports. Each ServiceEntry object corresponds to a single instance of a service. In one implementation, the ServiceEntry object provides descriptive and identifying properties including: (1) a service name; (2) a service description; (3) endpoints, which typically contain a network address(es) for the service; (4) a key, that identifies the service instance; (5) properties, *e.g.*, an extensible list of name-value pairs for service or device

characteristics; and (6) a provider, *e.g.*, an identifier that identifies the entity that provides the service.

[0042] A discovery persistence service 212 communicates with service discover API 214. Among other things, discovery persistence service 212 registers for announcement events over ad-hoc protocols. The discovery persistence service is notified when an announcement event is detected, and the discovery persistence service copies information about the service announcement into a memory location in data store 240. Storing service details in a memory location enables discovery of services that may be currently unavailable. For example, even if a printer is currently switched off details about the printer may be registered in the memory location and can be discovered. In addition, service queries are not restricted to the protocol that communicates with the service. Moreover, the performance of querying the memory location may be much better than issuing a broad network discovery query.

Exemplary Operations

[0043] In an exemplary implementation, the service discovery API 214 provides methods for service discovery, service publication, and subscribing to service event notifications. **Fig. 3** is a flowchart illustrating operations 300 for service discovery. At operation 310 an application defines a scope, at operation 315 the application defines a filter, and at operation 320 the application issues a

search request. The service discovery API 214 receives the search request and, at operation 325, the service discovery API 214 parses the search request. At optional operation 330, the service discovery API 214 determines whether the search request is resolvable using information stored in the discovery persistence service 212. In one implementation, information managed by the discovery persistence service 212 includes a time-of-life indicator that specifies the lifespan of the information in the discovery persistence service 212. Depending upon control and configuration, the service discovery API 214 may query the discovery persistence service 212 to determine whether the discovery request can be satisfied using information the discovery persistence service 212 manages on the data store 240. If the discovery request is resolvable using the discovery persistence service 212, then control passes to operation 350, and the service entry objects retrieved from the discovery persistence service 212 are returned to the application.

[0044] By contrast, if the discovery request is not resolved or resolvable using information managed by the discovery persistence service 212, then control passes to operation 335, and the service discovery API 214 executes the low-level API call(s) required to fulfill the discovery request. At operation 340 the service information returned from the low-level API calls is formatted into service entry objects, and at optional operation 345 the service entry objects are forwarded to the discovery persistence service, which may store the service entry objects on data store 240. At optional operation 347 further processing and filtering of the service

entry results such as duplicate detection and removal may be performed. At operation 350 the service entry objects are returned to the application for further processing, at operation 355. The particular details of the further processing performed by the application are not important.

[0045] **Fig. 4** is a flowchart illustrating operations for service publication. At operation 410 an application defines a service entry object for the service publication. At operation 415 the application defines the scope for the service publication. At operation 420 the application assigns a unique key to the service publication, and at operation 425 the application assigns a service type to the service publication. At operation 430 the application defines endpoints for the service publication, at operation 432 the application defines properties for the service publication and at operation 435 the application generates a publication request. The steps performed may vary according to the detail of information that is to be published and the low-level API that will be used.

[0046] The service discovery API 214 receives the publication request and, at operation 440, parses the publication request. At operation 450 the service discovery API 214 executes the low-level API calls to execute the service publication request. At optional operation 455 the service publication is stored in the discovery persistence service 212.

[0047] The service publication facilities of the service discovery API 214 can also be used to delete a published service. **Fig. 5** is a flowchart illustrating

operations for service deletion. At operation 510 an application defines a service entry object for the service publication. At operation 515 the application specifies the unique key for the service. At operation 520 the application defines a scope for the service deletion. At operation 530 the application generates a service deletion request.

[0048] The service discovery API 214 receives the deletion request and, at operation 540, parses the deletion request. At operation 550 the service discovery API 214 executes the low-level API calls to execute the service deletion request. At optional operation 555 the service publication is deleted from the discovery persistence service 212.

[0049] The service discovery API 214 can also be used to allow applications to be notified of service events, such as the arrival or departure of a new service or device of a particular type. **Fig. 6** is a flowchart illustrating operations 600 for subscribing to service events. At operation 610 an application defines a scope that specifies the particular low-level protocol to monitor. At operation 615 the application defines a filter that specifies the type of event. At operation 620 the application defines a callback function that will receive ServiceEntry details as matching events occur. At operation 625 an application generates a subscription request, which is forwarded to the service discovery API 214.

[0050] The service discovery request API 214 receives the subscription request and, at operation 630, parses the subscription request. At operation 635 the

service discovery request executes the low-level protocol calls required to implement the subscription service. When a service event occurs the low-level protocol will provide the service discovery API with a notification of the event. At operation 640 the event notification is formatted into a service entry object. At optional operation 645 the service entry object may be stored in the discovery persistence service 212, and at operation 650 the service entry object is returned to the application using the previously specified callback function. At operation 655 the application performs further processing on the service entry object. The particular details of the further processing performed by the application are not important.

[0051] The system's components and operations are discussed in greater detail below.

API Classes

Filters

[0052] A Filter is a set of rules by which a service description can be evaluated, resulting in true (*i.e.*, service description matches the filter) or false (*i.e.*, service description doesn't match the filter). A filter can be expressed either as a simple filter, which specifies particular properties, or as a rich filter, which uses more expressive grammar. Whether expressed as a simple filter or a rich filter, queries can be specified and executed over more than one protocol without

modification, subject to the capabilities of the underlying protocols. The service discovery request API 214 manages the re-expression of the higher level query into the correct format for the underlying low-level protocol. For example, the service discovery request API 214 can receive a query for a particular service type and express and evaluate it using LDAP for Active Directory and using the UDDI protocol for a UDDI Web service registry. An application developer is not required to work directly with the individual protocols.

[0053] In an exemplary implementation, the service discovery request API 214 requires discovery modules to support a simple filter, providing exact match semantics for provided criteria, and a rich filter containing a query expressed in the OPath grammar. It will be appreciated that each may also support additional "native" filter types. Different discovery modules may have protocol-specific native filter types, *e.g.*, UPnP may use XPath filters, Active Directory may natively use LDAP filters, and UDDI may natively use a UDDI filter.

[0054] The base level of OPath filter functionality across the modules further insulates applications from underlying discovery protocols. The filter class exposes additional methods to parse and interpret the filter in a way that is shared across the modules.

[0055] A simple filter provides for expression of queries by specifying a service type, services interfaces, and/or properties. Any combination of these

settings may be provided in a search query, and services will be included in the resulting service entry collection only if all of the criteria exactly match.

[0056] The service type may be implemented as a string that specifies the type that must match the service instances. A common set of service types are predefined in the service discovery request API 214. This set may be extended as key entities within protocols and stores are identified. For example, for printers in Active Directory, this would specify: filter.ServiceType = CommonServiceTypes.Printer.

[0057] The service interfaces may be implemented as a string collection that specifies identifiers for interfaces that services must match. As an example, for web services in UDDI, the following tModel identifiers could be specified: filter.ServiceInterfaces.Add("uuid:ac104dcc-d623-452f-88a7-f8acd94d9b2b"); filter.ServiceInterfaces.Add("uuid:4d2ac1ca-e234-142f-e217-4d9b2f8acd9b")

[0058] Properties may be implemented in a property dictionary that specifies service characteristics that services must match. As an example, for printers in Active Directory, the following properties could be specified: filter.Properties.Add ("printcolor", "TRUE"); filter.Properties.Add ("pagesperminute", "50")

[0059] A rich filter provides a mechanism for expressing significantly richer query semantics using, *e.g.*, the OPath grammar, by setting a Query string property. As an example, for web services in UDDI, the Query string would specify the required name and a required supported interface: filter.Query = "WebService[

name = 'Fabrikam' and ServiceInterface = 'uuid:ac104dcc-d623-452f-88a7-f8acd94d9b2b']"

[0060] As a more expressive example to find printers in Active Directory capable of printing more than 25 pages per minute where A4 paper is not available: filter.Query = "Printer[printPagesPerMinute > 20 + 5 and not(printmediaReady = 'A4')]".

[0061] Since the capabilities of the underlying protocols and stores are far from identical, ranging from the basic NetBT to the rich Active Directory query semantics, the ability to use the more expressive constructs of OPath will depend upon the scope (protocol) selected.

Scopes

[0062] A scope identifies a query domain that can be searched, usually coarse and by network location or administrative boundary. Discovery queries are directed to one or more scopes, and the query result includes a subset of the services within those scopes, *i.e.*, the query result is the subset of all services within the scope that match the given filter. Exemplary scopes include workgroup, localmachine, and domain.

[0063] The service discovery API 214 accommodates concrete scopes and abstract scopes. A concrete scope specifies a query domain in three pieces. A Protocol identifier that identifies a specific protocol, *e.g.*, mapping to a single

discovery module such as ConcreteScope.NetBtProtocol. or ConcreteScope.ADProtocol, an Address (optional) identifier that specifies a server to which to direct operations on this scope such as "http://intra-uddi/uddi/inquire.asmx" for an intranet UDDI server, and a path identifier (optional) that identifies a partition of the module's namespace, such as an LDAP search base which could be set to "CN=joe-dev,CN=Computers,DC=corp,DC=fabrikam,DC=com", or a UPnPv2 scope name.

[0064] The service discovery request API 214 passes concrete scopes to modules. The service discovery request API 214 does not preclude modules from performing additional indirection on concrete scopes such as, *e.g.*, transmitting the concrete scope over the wire to a second machine and passing the concrete scope to a corresponding API on that second machine.

[0065] An abstract scope is a moniker for one or more concrete scopes and possibly further abstract scopes. Abstract scopes provide a mechanism for targeting a query across a logical predefined or configured concrete scope collection. This provides an additional abstraction that allows the developer to target, for example, an "enterprise" scope, without requiring explicit protocol, address, and connection details for particular directory servers.

[0066] The mapping of abstract scopes to concrete scopes is machine-wide and configurable. For example, an abstract scope AbstractScope.Enterprise might map to include both of the concrete scopes in Table 1.

protocol = ConcreteScope.ADProtocol address = "ldap://dev.corp.fabrikam.com" path = null
protocol = ConcreteScope.UddiProtocol address = "http://uddi.fabrikam.com/inquire.asmx" path = null

Table 1

[0067] Fig. 7 is a block diagram illustrating an exemplary relationship between concrete scopes and abstract scopes. Concrete scopes 730-750 provide the specification of the domain across which queries will be evaluated. Concrete scopes 730-750 comprise protocol identification details and, as required, specifics of a store or server to use, with the potential for further scoping within that store or server. Within the service discover API 214, these are specified in the Protocol, Address and Path properties respectively.

[0068] Abstract scopes 710-725 provide a higher level hierarchical abstraction over and above concrete scopes. Abstract scopes are configured to include the concrete or abstract scopes that make them up. This scope mapping will be available to system administrators, who can be able to configure exactly how, for example, the AbstractScope.EnterpriseScope should be resolved.

[0069] Both concrete and abstract scopes can be used by a user of the service discovery API 214. In the case where an abstract scope is provided, the service discovery API 214 will resolve this down, through the hierarchy, to a number of concrete scopes.

[0070] Abstract scopes allow developers of application programs 210 to work at a relatively high level and include scope identifying terms such as "AbstractScope.Enterprise" in code. In this way, for example, the developer is not required to hardcode the specifics of a particular UDDI server into his code. This abstraction provides for greater reuse and portability of code. The same piece of code can be used in a variety of enterprise environments without change or recompilation. Only the abstract scope configuration would change between environments.

[0071] There may be multiple hierarchies of abstract to concrete scope mappings. In Fig. 7 AbstractScope.LocalMachine does not map up into AbstractScope.All even though all of its constituents are included.

[0072] In an exemplary implementation the scope map configuration may be manipulated through group policy by a system administrator to control the use of the service discover API 214 in the enterprise. By way of example, an administrator could define one or more abstract scopes available in the enterprise computing environment, or in a portion of the enterprise computing environment. This permits a system administrator to regulate the discovery and use of resources by applications.

ServiceEntry Results

[0073] An application developer can select appropriate Scope and Filter expression, which may then be set as properties on a service finder object. The application can then use the FindOne or FindAll methods to execute a discovery request. The FindAll method returns all services matching the supplied criteria, whereas the FindOne method returns a single matching service. The methods may be executed using a synchronous or an asynchronous calling pattern.

[0074] Assuming that there are services that match the provided filter within the specified scope, the FindOne or FindAll methods will return one, or a collection of, service entry objects. The service entry object is an abstraction over the various representations of services that the underlying protocols can provide. Each service entry object corresponds to a single instance of a service and as such, offers descriptive and identifying properties including those set forth in Table 2.

Property	Comments
Name	Identifies Service Instance
Description	Description of Service Instance
Endpoints	The set of endpoints at which the service instance can be accessed
Key	The identifying key for the service instance
Scopes	The scopes that an entity was discovered from or is to be published into
Credentials	Specifies the credentials that will be used when publishing this service.

Provider	References the "provider" (container) of the service, if any
Expiration	Time at which the service entry will expire, based on a time-to-live

Table 2

[0075] A public void Save() function is provided to create or update the service entry representation in the scopes specified in the scopes collection.

[0076] A public void Delete() method removes this ServiceEntry object from the scopes specified in the Scopes property. An exception will be thrown if the service is not already published.

Pseudo-Code

[0077] Figs. 8-24 illustrate pseudo-code for performing various service discovery, publication, and subscription functions.

[0078] Fig. 8 is pseudo-code illustrating how to use the C# programming language to locate color printers that print 50 pages per minute using a SimpleFilter object on the Active Directory protocol.

[0079] Fig. 9 is pseudo-code illustrating how to use the C# programming language to locate Web services that implement the uddi-org:inquiry_v2 interface and are named Fabrikam using the RichFilter object over the UDDI protocol.

[0080] Fig. 10 is pseudo-code illustrating the use of the C# programming language to find services supporting a specific tModel interface using a SimpleFilter object and the UDDI protocol.

[0081] Fig. 11 is pseudo-code illustrating the use of Visual Basic.NET to find services supporting a specific tModel interface using a SimpleFilter object and the UDDI protocol.

[0082] Fig. 12 is pseudo-code illustrating the use of the C# programming language to locate a printer with a name like Office Printer using the RichFilter with Active Directory.

[0083] Fig. 13 is pseudo-code illustrating the use of Visual Basic.NET to locate a printer with a name like Office Printer using the RichFilter with Active Directory.

[0084] Fig. 14 is pseudo-code illustrating the use of the C# programming language to publish a service of a specific type, identified by a specific unique identifier, using the SSDP protocol.

[0085] Fig. 15 is pseudo-code illustrating the use of Visual Basic.NET to publish a service of a specific type, identified by a specific unique identifier, using the SSDP protocol

[0086] Fig. 16 is pseudo-code illustrating the use of the C# programming language to delete a service from the SSDP protocol.

[0087] Fig. 17 is pseudo-code illustrating the use of Visual Basic.NET to delete a service from the SSDP protocol.

[0088] Fig. 18 is pseudo-code illustrating the use of the C# programming language to use a SimpleFilter to register for events of a specific type that use the

SSDP protocol. The registered callback function will be invoked for every event that matches the filter and the corresponding ServiceEntry object will be provided to that handler.

[0089] Fig. 19 is pseudo-code illustrating the use of Visual Basic.NET to use a SimpleFilter to register for events of a specific type that use the SSDP protocol.

Exemplary OPath Syntax

[0090] Table 3 provides exemplary OPath syntax for various discovery functions.

OPath	Refers to
Printer	Find all printers and print queues.
Printer[name = 'Upstairs Printer']	Find all printers where the name is Upstairs Printer.
Printer[printPagesPerMinute > 20 + 5 and not(printmediaReady = 'A4')]	Find all printers capable of printing more than 25 pages per minute and A4 paper is not available.
Printer[Properties.name like 'Pri' and (printPagesPerMinute > 10 or printMediaReady = 'letter')]	Find all printers where the name begins with Pri and either the pages per minute is greater than 10 or letter paper is available.
Printer[supportsColor = true && (printerName like 'Home' or name like 'Work')]	Find all printers which support color and the name begins with Home or Work.
Service[ServiceInterface=ServiceConnectionPoint]	Find all services which are ServiceConnectionPoint objects.

Service[(serviceType = 'Printer' or serviceType= 'Computer') and name like 'Work']	Find all services, either printers or computers, that have a name like Work.
Computer[operatingSystemVersion like '%3790%']	Find all computers that are running an operating system whose version number contains 3790.
Computer[operatingSystem='Windows Server 2003']	Find all computers that are running a particular operating system. The **operatingSystem** attribute is not included in the global catalog.

Table 3

[0091] Table 4 contains examples of OPath syntax that can be used on the UDDI protocol.

OPath	Refers to
WebService[name = 'Fabrikam']	Find all Web services where the name is Fabrikam.
WebService[name = 'UDDI%' && ServiceInterface = 'uuid:ac104dcc-d623-452f-88a7-f8acd94d9b2b']	Find all Web services where the name starts with UDDI and that supports the identified interface (i.e. the tModel uddi-org:inquiry_v2).

Table 4

[0092] Table 5 contains examples of OPath syntax that can be used on the NetBT protocol.

OPath	Refers to
Workstation	Find all workstations.
Service[ServiceType = 'PrintQueueServer']	Find all services of type PrintQueueServer.
Computer[serviceInterface = 'DomainController' and ServiceInterface = 'TerminalServer']	Find all domain controller computers running as a terminal server.

Table 5

Discovery Persistence Service

[0093] As described briefly above, the discovery persistence service 212 manages a persistent data store for service information. Periodically, or at predetermined events, such as startup, the discovery persistence service registers to receive ad-hoc device/service announcements. As an example, when a new UPnP device is introduced it will generate a device announcement that will be handled by the UPnP protocol module. This module will then surface details of that event (the device and its services) to the discovery persistence service through the service discovery API 214.

[0094] Using its persistent data store, the discovery persistence service then determines whether this is a new device/service or a returning device/service. If it is a new device/service, the details of the device and its services will be registered in the persistent data store. When another consumer of the service discovery API 214 then attempts to find services, the service discovery API 214 will be able to

return services for ad-hoc devices/services, even if the devices are not currently available. For the above example, in the case where the device/service is currently available, depending upon the scope specified, both the UPnP protocol module and the persistent data store module may return results for the device. In addition to UPnP, this functionality applies to other ad-hoc discovery mechanisms.

[0095] Thus, the discovery persistence service 212, the service discovery API 214, and the local database store 240 provide a layer of abstraction over the various low-level protocols used for device and service discovery. This additional layer of abstraction establishes a common and improved search semantic that application developers may use in developing applications.

[0096] In addition, the discovery persistence service 212, the service discovery API 214, and the local database store 240 provide a consolidated discovery model for services and devices on a local machine, a home network(s), an enterprise network(s), and the internet. Thus, application developers can discover services in a wide variety of locations by writing to a single, consistent API.

Conclusion

[0097] Although the described arrangements have been described in language specific to structural features and/or methodological operations, it is to be understood that the subject matter defined in the appended claims is not

necessarily limited to the specific features or operations described. Rather, the specific features and operations are disclosed as preferred forms of implementing the claimed present subject matter.